Introduction to the HTTP Server

This chapter tells you how the HTTP server works. It discusses the SMARTS implementation of the CGI interface for C programs; the SMARTS extensions to CGI for 3GL languages and Natural; and the SMARTS conversational CGI program concept.

This chapter covers the following topics:

- The HTTP Server
- SMARTS Implementation of the Common Gateway Interface (CGI)
- SMARTS CGI Extensions
- The Conversational CGI Program Concept

The HTTP Server

The HTTP server, based on the HTTP version 1.1 standard, is built on the capabilities of the SMARTS API and the Software AG Com-plete system. It can form the heart of a worldwide web site on the mainframe operating system and provides the vital link between the mainframe resident data and a company intranet or internet web site.

The HTTP server has the following highlights:

- Any form of data including static HTML pages, GIF files, JAVA classes can be delivered from the mainframe host to an HTTP user. Users of browsers on PCs or UNIX systems can access data on the mainframe.
- For text-based data, the HTTP server handles all EBCDIC/ASCII issues using configurable translation tables.
- Common Gateway Interface (CGI) support is fully integrated with the SMARTS API so that C CGI programs that comply with the CGI ‘standard’ can run.
- Extensions provided by the HTTP server make it possible for Natural, COBOL and PL/1 programs to operate as the target of CGI requests.
- When SAF security checking is active and the HTTP server is running under Com-plete, the HTTP server interfaces fully with three different modes of operation from running all users with the same SAF authorization to a situation where each and every user must provide a user ID and password to the HTTP server before any requests are serviced.
- When running in the Com-plete environment, one or more HTTP servers listening on different ports can be operational within the same SMARTS.
- Each HTTP server may operate with a different security mode.

SMARTS Implementation of the Common Gateway Interface (CGI)

Note:
General information about the Common Gateway Interface (CGI) protocol is available on the Internet. You can use any search engine available on the Internet to find the CGI information.

SMARTS CGI Input Processing

CGI input is processed according to the standard and depends on the HTTP request method used to invoke the CGI program:

- When the ‘GET’ HTTP request method is used, the input parameters are provided in the QUERY_STRING environment variable and any attempt to access stdin results in an ‘end of file’ condition being raised.
- When the ‘POST’ HTTP request method is used, the input parameters are provided as a stdin stream and may be read using any valid SMARTS function to read stdin input. The amount of input to be expected may be determined from the CONTENT_LENGTH environment variable.

Input Translation

All data from the web browser is submitted in standard 7-bit ASCII codes. The HTTP server translates to EBCDIC and sets up headers in standard EBCDIC format as appropriate.

- When the input content type, as specified by the CONTENT-TYPE HTTP header, indicates that the input is type TEXT, the input is translated from ASCII to EBCDIC and is made available to the application program in EBCDIC format.
- When the input content type is TEXT/X-WWW-URLENCODED, any hex values in the data as specified by %xx where ‘xx’ is the ASCII hex code for the character being submitted to the CGI request, have the ‘xx’ translated to the appropriate EBCDIC hex representation of that character. The application may then safely convert this value to a character and be sure that it will be the desired EBCDIC equivalent of that character.

  For example, the equals character ‘=’ has an ASCII representation of X‘3D’ while it has an EBCDIC representation of X‘7E’. In the TEXT/X-WWW-URLENCODED stream, it appears as ‘%3D’. The HTTP server processing converts this to ‘%7E’ so that the user program sees the EBCDIC representation of the character.

SMARTS CGI Output Processing

CGI output is sent to the stdout stream using any standard C functions normally used to send output to stdout.

Header Processing

The application may provide all, some, or none of the HTTP headers, which provides the greatest flexibility in responding to requests. Two forms of output headers can be supplied:

- nonparsed header (NPH) output; and
- parsed header output.

Nonparsed Header Output

Nonparsed header (NPH) output is supported. It is identified by the string ‘HTTP’ sent as the first four characters of output. The SMARTS CGI processing routines simply pass all data to the browser directly from the CGI application, so the CGI application can send any header it wishes and any data it likes. The CGI program is then completely responsible for accuracy.

Note:
If a CGI application sends all HTTP headers itself, conversational processing is not supported for the HTTP server.

Parsed Header Output

In most cases, a user CGI program provides a single HTTP header: the CONTENT-TYPE. This is sent as the first string in response to a CGI request and is followed by two CRLF sequences indicating that the data follows (headers are followed by only one CRLF sequence). Optionally, the application may include more than the CONTENT-TYPE header before sending the CRLFCRLF sequence followed by data.

If SMARTS does not detect a CONTENT-TYPE header in the first output from the CGI program, it includes a CONTENT-TYPE header using the default content type for the HTTP server processing the request. Following the CONTENT-TYPE header, the user may submit zero or more HTTP headers.

Alternatively, the LOCATION header may be sent as the first (and only) header. No CONTENT-TYPE header is sent and the LOCATION header is handled according to the HTTP protocol.

Providing No HTTP Headers

When a program sends no HTTP headers at all, which happens if an application was written to stdout in a number of environments, the HTTP server inserts a default CONTENT-TYPE header as specified in the DFLCONT configuration parameter. Software AG recommends that you set this parameter to TEXT/PLAIN to accommodate all programs. Other content types may cause confusing results.

Output Translation Processing

All HTTP headers, whether generated by SMARTS or the CGI program, are expected in EBCDIC and are translated by the HTTP server to ASCII prior to being sent to the CGI requester. This ensures that application programs have readable headers included instead of setting up ASCII data streams using EBCDIC tools and editors.

SMARTS also monitors the data stream for the start of the content itself, which is signified by a CRLFCRLF sequence in the output data stream (headers are followed by only one CRLF sequence). Once the CRLFCRLF sequence is detected, all other data is treated as output content.

SMARTS translates output content from EBCDIC to ASCII only if the output type sent by the CGI program, or defaulted by the HTTP server, is of type TEXT/*. This includes TEXT/PLAIN, TEXT/HTML but also occurs for any other user TEXT/x-application-* type content headers.

Any other content types are sent through unchanged to the requester of the CGI program. This facilitates downloading binary data of any sort.

SMARTS CGI Environment Variables

The following table summarizes the environment variables and their contents set up as a result of a CGI request processed by SMARTS:

Environment Variable	Contains ...
REQUEST_METHOD	a value indicating the HTTP request method used to generate the request. Normally this is GET or POST and indicates where the input date, if any, can be found.
SERVER_PROTOCOL	the version level of the HTTP protocol used to send the request to the server. Normally this has the form HTTP Vv.r where ‘v’ and ‘r’ indicate the level of HTTP being used.
SERVER_PORT	the HTTP port number being used to service requests by this HTTP server.
CONTENT_LENGTH	the length of the input data available on the stdin stream for processing CGI requests generated by the POST request method.
CONTENT_TYPE	the type of the input data available on the stdin stream for processing CGI requests generated by the POST request method.
QUERY_STRING	the parameters provided with a content type of TEXT/X-WWW-URLENCODED for CGI requests generated by the GET request method.
PATH_INFO	the piece of the URL used to invoke the CGI program following the CGI program name.
PATH_TRANSLATED	the physical path used to run the CGI program, which may differ on some systems from the URL path. The translated path is always the same as the URL path when running under the HTTP server.
SCRIPT_NAME	the piece of the URL up to the end of the CGI program name. SCRIPT_NAME and PATH_INFO together compose the URL.
REMOTE_USER	the remote user ID provided with the HTTP request.

In addition to the above, all HTTP headers are set up as environment variables by prefixing them with the string ‘HTTP_’ and translating all ‘-’ or dash characters in the HTTP header name to ‘_’ or underscore characters as per the CGI standard. For example, the header ‘LOCATION’ may be obtained by requesting ‘HTTP_LOCATION’ if this header was provided on the request.

Note:
All HTTP environment variable names are uppercase EBCDIC values.

SMARTS CGI Termination Processing

When a CGI program terminates normally, it sends all output data to the stdout stream and returns control to the HTTP server. The HTTP server ensures that all data is sent to the web browser and then closes the connection.

When the HTTP client browser supports persistent sessions, the data is sent to the browser but instead of closing the connection, the HTTP server waits for another request over the same connection.

When a CGI program terminates abnormally either by ABENDing or not returning control to the HTTP server, the web browser may receive some or none of the data sent by the CGI program. In this case, the HTTP server's priority is to clean up the session by closing the connection and freeing all resources associated with that request, thus preventing resource 'leaks' that would otherwise impact the integrity of the HTTP server later in its cycle.

The HTTP server configuration parameter SEND determines the amount of data seen:

- If SEND=IMMEDIATE is set, the client browser sees all data written to stdout to the point where the program ABENDs. Software AG recommends that you set this option in a test environment.
- If SEND=BUFFERED is set, the client browser only sees data if the buffer was filled at least once prior to the ABEND. All data in the buffer at the time of the ABEND is lost. Software AG recommends that you set this option in a production environment for greater performance.

SMARTS CGI Extensions

Now that you understand how a CGI request is generated and processed by a CGI program, this section describes in general terms how languages use the SMARTS CGI extensions. The language-specific chapters provide additional detail.

Standard CGI Operation

CGI is essentially a remote procedure call (RPC) type request driven by an HTML page that results in a request being sent to the HTTP server to run a specific program. The request includes data from the browser filled in by the user of the HTML page that generated the request.

The HTTP server

- recognizes the CGI request;
- makes information from the submitted HTML page and information about the actual request available in the execution environment of the CGI application; and
- gives control to the program identified by the request.

  Information is made available

- by using the C stdin stream for certain types of input; and
- by setting up well known environment variables that may be accessed using the getenv function.

By writing to the stdout stream, output is submitted back to the requester with the response.

Non-C CGI Programs

The CGI standard was designed specifically for the C language; however, many installations need to leverage their current skills by providing CGI support in other languages.

Since most languages cannot take advantage of the ‘stream’ approach for accessing data used by the CGI interface, SMARTS provides extensions to the standard to enable languages other than C to access ‘input’ from a web CGI request and produce output in response to that input. The SMARTS extensions make it possible to write CGI routines in languages that are as powerful and productive as their C counterparts.

Extensions for Other Languages

Languages such as Natural and COBOL are not suited to interpreting streams of data or parameters provided in strings. Implementation in PL/1 is clumsy while Assembler processing of this data is long-winded and time-consuming from a coding point of view.

Rather than adapt other languages to the C way of doing things, SMARTS provides CGI extensions for accessing data submitted in a CGI request and building the response to that request. The CGI extensions are implemented using a simple call mechanism that allows applications to be written in a more structured mode than normal CGI programs; a mode that suits languages like Natural, COBOL, and PL/1.

Every piece of data submitted as part of a CGI request has a field name associated with it. Where data can be entered in a field, the user enters the field name and the data. As CGI programs are designed to respond to the input from a given HTML page, or at a minimum, HTML pages with certain fields defined, there is no need for search strings.

The SMARTS primary user program request interface is called HAANUPR. This interface will be maintained and upgraded in the future.

For compatibility, previous interface modules continue to be supported so that existing applications can run unchanged, but they will not be enhanced.

These interface modules are documented in detail in the chapter Programming CGI Requests in this manual.

The Conversational CGI Program Concept

Since the foundation of the worldwide web, all CGI programming has been non-conversational or "stateless". This means that when a request is issued, the CGI program processes the request, returns the response to the client, and forgets all about that client.

Some CGI applications store information away about the user’s state and retrieve it when the user appears again; however, the state information must then be aged as the user may never come back. It also complicates the program in terms of transactional processing over more than one CGI request, identification of users, and so on.

The HTTP Server Solution

The HTTP server conversational concept avoids these problems by enabling an application program to issue a CONVERSE call in the middle of a processing loop. This sends all data previously output by the CGI program to the client browser.

The HTTP server then suspends the application program until the next HTTP request is issued for that user.

When it is received, the appropriate session is restarted by the HTTP server after the call to CONVERSE with all the data from the client browser available to it as well as any data, switches, or database sessions that the CGI program had in local storage prior to the CONVERSE.

It is possible to set timeouts so that the session context is cancelled if a user doesn’t respond within a specific period of time (for example, 30 minutes). When the user does respond, it receives a message to the effect that the conversation doesn’t exist.

This mechanism has the following advantages:

- A conversational program can save valuable resources as it is only started once per user and remains active until the conversation ends. Normally, a CGI program must be started and terminated for every CGI request.
- Programs can be structured properly with standard loop and parameter gathering techniques.
- The application program can maintain database or any other connections over the duration of the conversation, avoiding the need to connect and disconnect for each CGI request.
- It gives a CGI application the ability to maintain a transaction over a conversation of two or more HTML pages, if necessary.
- It is a more natural way to program instead of having to collect all the context information which will be required the next time input appears from a given user.

Conversational CGI Application Structure

Any CGI program may converse if the server where the CGI program will run is configured to allow conversational programs. This is controlled by the HTTP server CONV configuration parameter, which must be set to YES to allow conversational programs. Because the CGI program uses the ENABLE-CONVERSE, CONVERSE, and DISABLE-CONVERSE functions of the HAANUPR interface, this facility may be used from any programming language including Natural, COBOL, PL/1, and Assembler.

The following pseudo-code illustrates how such an application is structured. The member HOANCONV on the HTPvrs.SOURCE dataset is an example of a conversational COBOL CGI program. Comments in the following are enclosed using "/*" to start the comment and "*/" to terminate the comment.

```
BEGIN:
/*
   The following call tells the SMARTS HTTP server that you wish to have a
   conversation with the client browser. If conversations are not supported,
   this call will fail
*/
CALL ‘HAANUPR’ status ‘ENABLE-CONVERSE’
```

```
/*
   In the following logic, the program fields would be filled out with the
   initial values to be presented to the client browser.
*/
Set initial values in output HTML page
/*
   The program will always loop indefinitely until the client at the browser
   indicates that it wants to terminate the conversation, or the program itself
   decides to terminate. The client may indicate this by entering a certain
   value in a field or by pressing a specific HTML button. The program may do
   this based on a transaction being completed.
*/
Do while not end of conversation
/*
    The HTML page is built using multiple calls to the HAANUPR interface to
    output the appropriate HTML to the client. This may be preceded by any
    other logic to get data from a database or build data based on time or
    whatever.
*/
CALL ‘HAANUPR’ status ‘PUT-TEXT’ data length
CALL ‘HAANUPR’ status ‘PUT-TEXT’ data length
..
/*
   The only requirement in terms of output is that the CGI program must ensure
   that it gets control back by using the appropriate URL on its SUBMIT
   buttons. The easiest way is to use a relative URL like ‘/cgi/program/’
   where ‘program’ is the name of the CGI program. The CGI program is then
   redispatched then next time the SUBMIT button is pressed. If the wrong URL
   is used in this area, the conversational CGI program is never redispatched.
*/
CALL ‘HAANUPR’ status ‘PUT-TEXT’ data length
```

The following is the end of the DO WHILE loop:

```
*/
END
/*
```

Once you arrive here, the conversation may be terminated:

```
*/
CALL ‘HAANUPR’ status ‘DISABLE-CONVERSE’
/*
     Write a final ‘goodbye’ message to the user
*/
CALL ‘HAANUPR’ status ‘PUT-TEXT’ data length
END
```